Amendments to the Specification

Please replace paragraph **[0002]** with the following amended paragraph:

Certain software technologies, such a ~~Java~~ Java™, emphasize the use of a special interpreter (which may also be referred to as a "virtual machine") that allows generic processor instructions to be executed on a particular type of processor. Here, each hardware platform (e.g., each computer) that the generic instructions are expected to "run on" typically includes a virtual machine that is responsible for converting the generic processor instructions (which are typically referred to as "bytecode") into instructions that are specially targeted for the hardware platform's particular processor(s). Software technologies that embrace the execution of bytecode on a virtual machine may be referred to as "virtual machine-based" software.

Please replace paragraph **[0003]** with the following amended paragraph:

As a classic instance of the benefit of the ~~Java~~ Java™ virtual machine with respect to Internet usage, a first ~~PC~~ Personal Computer (PC) that is powered by an Intel processor may download from the Internet the same Java bytecode instructions as a second PC that is powered by a PowerPC™ processor. Here, the first PC's ~~Java~~ Java™ virtual machine converts the ~~Java~~ Java™ bytecode into instructions that are specific to an Intel processor while the

second PC's ~~Java~~ Java™ virtual machine converts the same ~~Java~~ Java™ bytecode into instructions that are specific to a PowerPC™ processor. Thus, through the use of ~~Java~~ Java™ bytecode and processor specific ~~Java~~ Java™ virtual machines, an Internet server is able to maintain only a single type of code (the ~~Java~~ Java™ bytecode) without concern of client compatibility.

Please replace paragraph **[0005]** with the following amended paragraph:

Certain software technologies, including ~~Java~~ Java™, are "object oriented." According to an object oriented approach, the subject matter that is processed by a computer program is organized into classes of likeness. For example, the software used to sell items to customer X might belong to the same class of software (e.g., a class named "sales") that is used to sell items to customer Y. Here, given that a significant degree of overlap is expected to exist regarding the methods and data types used to process sales for both customers X and Y (e.g., an "update billing about sale" method, an "update accounting about sale" method, a "part number" data type, a "quantity" data type . . . etc.) it is deemed more efficient to organize such methods and data types into a generic "sales" class from which specific instances of the class (e.g., an instance for selling to customer X and an instance for selling to customer Y) can be defined and created.

Please replace paragraph **[0010]** with the following amended paragraph:

Over the course of discussion of various inventive aspects set forth in the detailed description that follows, comparisons will be made against each of **Figures 1c** and **1d**. **Figure 1c** illustrates, in more detail, exemplary ~~Java~~ Java™ source code level commands 120a and corresponding ~~Java~~ Java™ bytecode level instructions 120b for the "GetMax" method that was first presented in **Figure 1b**. The "GetMax" method is designed to return the greater of two variables 'a' and 'b' (i.e., if 'a' is greater, 'a' is returned; if 'b' is greater, 'b' is returned).

Please replace paragraph **[0011]** with the following amended paragraph:

Note that both the source code level and bytecode level implementations for the "GetMax" method have a single entry point 140a, 140b (i.e., the method starts at locations 140a, 140b) and a pair of exit points 141a, 142a and 141b, 142b (i.e., the method can end at locations 141a, 142a and 141b, 142b – noting that a "return" command/instruction causes an output to be presented; which, in turn, can be viewed as the completion of the method). Those of ordinary skill will be able to recognize that: (1) the source code level depiction 120a of the GetMax method observed in **Figure 1c** articulates a method written in ~~Java~~ Java™ source code language that returns the greater of two values (i.e., a and b); and, likewise, (2) the bytecode level depiction 120b of the GetMax method observed

in **Figure 1c** articulates a corresponding method written in ~~Java~~ Java™ bytecode language that returns the greater of two values (i.e., the values a and b which are respectively loaded on the top of an operand stack by the initial "iload_0" and "iload_1" instructions).

Please replace paragraph **[0020]** with the following amended paragraph:

The multi-tiered architecture illustrated in **Figure 2b** may be implemented using a variety of different object-oriented application technologies at each of the layers of the multi-tiered architecture, including those based on the ~~Java~~ Java™ 2 Enterprise Edition™ ("J2EE"). In a J2EE environment, the business layer 244, which handles the core business logic of the application, is comprised of Enterprise Java Bean ("EJB") components with support for EJB containers. Within a J2EE environment, the presentation layer 242 is responsible for generating servlets and ~~Java~~ Java™ Server Pages ("JSP") interpretable by browsers at the user interface layer 240 (e.g., browsers with integrated ~~Java~~ Java™ virtual machines).

Please replace paragraph **[0052]** with the following amended paragraph:

Figures 3 and **Figure 4a-b** describe techniques that can be directed to the testing, debugging and/or monitoring of sophisticated object-oriented virtual machine-based software. Throughout the description, for the

purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the present invention may be practiced without some of these specific details. For example, while the embodiments described below focus on a ~~Java~~ Java™ environment in which ~~Java~~ Java™ "bytecode" is processed by a ~~Java~~ Java™ "virtual machine," various underlying principles may be implemented in interpreted-code and non-interpreted-code environments as well as object oriented and non-object oriented environments.

Please replace paragraph **[0061]** with the following amended paragraph:

Figure 4a shows one embodiment of a compilation and execution methodology for the bytecode modification strategy outlined above. According to the methodology of **Figure 4a**, the source code of an object-oriented virtual machine-based software technology (e.g., ~~Java~~ Java™) is compiled 451 into its corresponding bytecode. The bytecode is then modified 452 by inserting additional bytecode instructions at method entry and exit points. The additional bytecode instructions invoke (e.g., make a function call to) a dispatch unit such as the dispatch unit 330, 430 illustrated in **Figures 3** and **4b.**

Please replace paragraph **[0076]** with the following amended paragraph:

Comparing the pre-modification bytecode method 120b of **Figure 1c** with the post modification bytecode method 620b of **Figure 6a**, note that additional

blocks of instructions 643b, 644b, 645b and 646b have been introduced by the bytecode modification process 452. Recalling that the modification process involves adding instructions that invoke the dispatch unit 430, and that the code observed in **Figure 1c** and **Figure 6a** are written in the ~~Java~~ Java™ language, each of the additional blocks of instructions 643b, 644b, 645b and 646b correspond to blocks of bytecode-level instructions that are designed to at least invoke the dispatch unit 430.

Please replace paragraph **[0077]** with the following amended paragraph:

Figure 6a, as an example, shows an "invokestatic" instruction being associated with each block of additional instructions. In the ~~Java~~ Java™ bytecode language, the "invokestatic" instruction is used to invoke a static method of a particular class. A static method is a method that belongs to a particular class but does not require an object of the class to be called upon in order for the method to be executed. Accordingly, use of the invokestatic instruction suggests that the dispatch unit 430 of **Figure 4** may be constructed as a class (e.g., a "dispatch" class) having static methods that are invoked by the modified methods. Note that other approaches are possible so that the dispatch unit methods need not be static methods while still complying with the underlying principles of the invention. For example, the invoked methods may be

associated with called upon objects (in which case, for ~~Java~~ Java™ applications, "invokevirtual" or "invokespecial" invoking instructions may be used).

Please replace paragraph **[0115]** with the following amended paragraph:

Thus, in contrast to the application tracing plugin 810 which causes the bytecode modifier 452 to modify all of the methods within a particular application, the user-configurable plugin 820 illustrated in **Figure 8** provides a finer level of granularity for tracing program flow. An "application" may be built from a plurality of packages (typically *.jar files in a ~~Java~~ Java™ environment); each package may be built from a plurality of classes (i.e., class files); and each class include a plurality of methods. As indicated in **Figure 8**, the user-configurable plugin 810 allows the end-user to identify specific packages, classes and/or individual methods to be modified by the bytecode modifier 452, thereby providing significantly greater precision for tracing and debugging operations. By way of example, if a coding problem is isolated to within a specific package, then only that package need be modified. Similarly, if the problem can be isolated to within a particular class or method, then only that class/method need be modified. In one embodiment, the different packages, classes and/or methods are selected and modified via one of the interfaces described below with respect to **Figures 19a-e**.

Please replace paragraph **[0118]** with the following amended paragraph:

Figure 10a shows an exemplary system comprised of an application server 1010 and a database server 1020. Application components 1011 executed on the application server 1010 may include, by way of example, presentation logic and business logic components. Within a J2EE environment, the presentation logic may contain servlets and Java Server Pages ("JSP") interpretable by browsers on clients 1000 (JSPs/Servlets generate HTML which is interpreted by browsers), and the business logic may include Enterprise ~~Java~~ Java™ Bean ("EJB") components which are supported by EJB containers. However, as previously mentioned, the underlying principles of the invention are not limited to a Java implementation.

Please replace paragraph **[0123]** with the following amended paragraph:

Figures 11a-b illustrate another embodiment of the invention in which distributed statistical information is collected using the bytecode modification techniques described herein. In **Figure 11a-b**, the application components 1111 within the application server 1110 communicate with and/or exchange data with an external system 1120 (e.g., such as an R3 system designed by SAP AG) as opposed to a database server as in **Figures 10a-b**. However, similar principles apply. In this embodiment, ~~For~~ for example, the client's 1100's initial request triggers a first method invocation 1101 within the application components 1111. The request may be, for example, a request for a particular record within the database. After additional application-layer processing, the application

components 1111 generate an external request via method invocation 1103 which is received and processed by the external system 1130. The specific format used for the external request may be based on the type of communication protocol supported by the external system 1130. Possible communication formats include (but are not limited to) remote method invocations ("RMI") or remote function calls ("RFC"). RMI is a type of remote procedure call which allows distributed objects written in Java (e.g., ~~Java~~ Java™ services/components) to be run remotely. RFC is a communications interface which allows external applications to communicate with R/3 systems. It should be noted, however, that the underlying principles of the invention are not limited to any particular communications protocol for communicating with an external system 1130.

Please replace paragraph **[0126]** with the following amended paragraph:

Figure 12 illustrates one embodiment of the invention in which the bytecode modifier 452 modifies a particular set of methods 1211-1232 to trace program flow within a J2EE engine 1250. For example, HTTP requests transmitted by a Web-based client 1200 (e.g., a client with a Web browser) are processed by input/output method invocations 1227-1228 of HTTP logic 1201. Communication between HTTP logic 1201 and servlet/JSP components 1202 is accomplished via method invocations 1229, 1230 and 1231, 1232, respectively. Other method invocations illustrated in **Figure 12** which represent entry/exit

points between different J2EE services include method invocations 1213, 1214, 1216 and 1215 between enterprise ~~Java~~ Java™ bean ("EJB") components 1203 and servlet/JSP components 1202; method invocations 1233, 1234 and 1219, 1220 between servlet/JSP components 1202 and Java connector ("JCo") components 1204; method invocations 1221 and 1222 between JCo components 1204 and an external system 1207; method invocations 1211, 1212 and 1223, 1224 between servlet/JSP components 1202 and ~~Java~~ Java™ database connectivity ("JDBC") components 1205; and method invocations 1225, and 1226 between the JDBC components and a database 1206; and method invocations 1217 and 1218 between EJB components and a non-web client (e.g., via RMI or RFC transactions).

Please replace paragraph **[0127]** with the following amended paragraph:

The specific functions performed by each of the modules/components illustrated in **Figure 12** are well defined and are not necessary for an understanding of the underlying principles of the present invention. For example, the JDBC component 1205 is a well known interface that allows Java applications to access a database via the structured query language ("SQL"). The JCo component 1204 is an interface that allows a Java application to communicate with systems designed by SAP AG (e.g., an R/3 system). As described above, HTTP logic 1201 and servlet/JSP components 1202 perform various well defined presentation-layer functions and the EJB components 1203 perform various well define business layer functions. Further details related to

each of these modules/ components can be found from various sources including the ~~Java~~ Java™ website (see, e.g., http://java.sun.com/).

Please replace paragraph **[0160]** with the following amended paragraph:

As illustrated in **Figures 19a** and **g**, in one embodiment, extensions 1903 are provided to the Apache Ant builder application 1902. Apache Ant is a well known open-source build tool for compiling, packaging, copying, etc. – in one word "building" – ~~Java~~ Java™ applications (see, e.g., http://jakarta.apache.org/ant). The extensions 1903 to the Ant builder application 1902 allow the end user to generate modified bytecode as part of the build process. For example, an ant task may be particularly suitable for automatic build scripts in which it is used to modify one or more classes after compilation (e.g., within a J2EE engine).